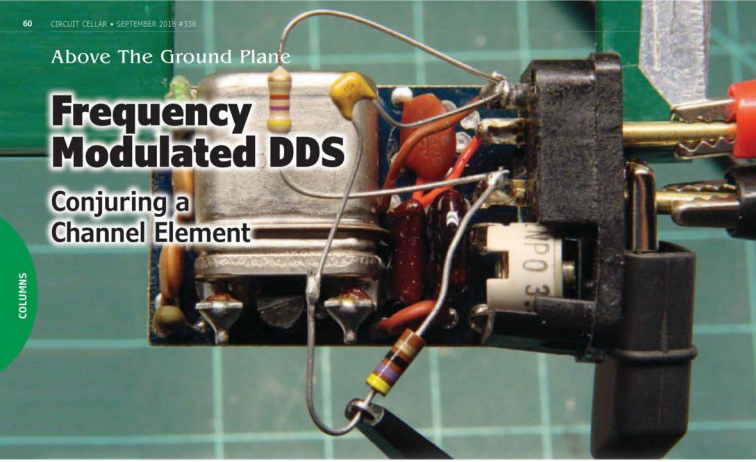
Above The Ground Plane

Frequency Modulated DDS

Conjuring a Channel Element



Prompted by a reader's query, Ed became aware that you can no longer get crystal oscillator modules tuned to specific frequencies. With that in mind, Ed set out to build a "Channel Element" replacement around a Teensy 3.6 board and a DDS module.

By
Ed Nisley

The 60 kHz LF Crystal Tester I described last year (Novemeber 2017, *Circuit Cellar* 328) prompted a reader to ask whether a similar DDS-based device could produce a frequency modulated signal from an audio input. He converts surplus commercial radio transceivers into duplex amateur radio repeaters and can no longer get crystal oscillator modules tuned to specific frequencies, because all the small companies formerly cutting crystals have vanished.

The module in the lead photo above, known as a Channel Element (CE), contains a temperature compensated crystal oscillator with a 5 ppm tolerance. I removed the CE's aluminum shield and plastic insulator, then soldered enough parts in place to run the oscillator. A typical Motorola MICOR radio might use four or eight CEs to set its receive frequencies, plus another group for the corresponding transmit frequencies. Fortunately, an amateur repeater requires only a pair of CEs to set the receive and transmit frequencies of its single "channel."

Over the years, different radios used a bewildering variety of CE sizes, shapes, frequencies and tolerances. For example, the CE shown here oscillated at 13.466666 MHz, the radio tripled the CE frequency to 40.400 MHz, then applied a +5.26 MHz IF to set a 45.660 MHz receive frequency. Although the frequency multiplication ratios range from 2 through 36, with a myriad IF offsets and signs, all the CEs oscillate between 4 MHz and 25 MHz—well within the range of inexpensive DDS modules similar to the Analog Devices AD9850 board in my Crystal Tester.

A 45.660 MHz signal lies in what's called the "Low Band" of commercial services, just below the amateur radio band starting at 50 MHz. A Low Band radio modified for amateur service at 51.140 MHz would need a CE crystal cut for 15.293333 MHz—not an item found on Amazon or Aliexpress.

CEs in the transmit chain included additional circuitry to vary the crystal frequency based on the microphone audio, but, unlike the broad signal from the local HD FM rock station I showed in my May 2017 article (*Circuit Cellar* 322), a 3 kHz audio bandwidth suffices for these radios. Sampling the audio well above the Nyquist rate with a microcontroller's built-in ADC seemed feasible, so I built the prototype "CE replacement" shown in **Figure 1** around a PJRC Teensy 3.6 and an Analog Devices AD9851 DDS module.

In this article, I'll explain how the Teensy's 32-bit datapath and 180 MHz CPU clock affect the DDS frequency calculations, then explore some detailed timings. The hardware sprawled across the proto board obviously won't fit into the same aluminum shield as the original CE, but it's a start!

FREQUENCY MODULATION

A frequency modulated signal starts with a constant-amplitude sinusoidal waveform at a constant "carrier" frequency:

$$x_c = A_c \cos(2\pi f_c t)$$

The simplest possible modulating signal would be another sinusoid at a different frequency:

$$x_m = A_m \cos(2\pi f_m t)$$

A frequency modulation circuit converts the amplitude of x_m into a corresponding frequency deviation f_Δ, with the ratio between the two given by a constant K_f:

$$K_f = \frac{f_\Delta}{A_m}$$

For example, if a 1 V signal produces a 5 kHz frequency deviation, K_f = 5 kHz/V. Note that the deviation depends only on the modulating signal's amplitude, not its frequency.

Even using just one fixed-frequency sinusoid as the modulating signal, a closed-form solution requires Bessel functions and math well beyond the scope of this article. Although my EE courses introduced me to Mr. Bessel, we haven't been on speaking terms ever since.

Several useful values do emerge from the math, however, including the carrier frequency deviation Δf and the modulation index β:

$$\Delta f = \max(f_\Delta)$$

$$\beta = \frac{\Delta f}{f_m}$$

Feeding β into the Bessel function produces the amplitudes of the sidebands on either side of the carrier due to the modulating signal, but, for most practical purposes, Carson's Rule gives the bandwidth required for a given modulating signal and modulation index:

$$BW = 2f_m(\beta + 1)$$

Amateur radio repeaters generally use a frequency deviation Δf = 5 kHz with an audio bandwidth f_m = 3 kHz:

$$\beta = \frac{5\,\text{kHz}}{3\,\text{kHz}} = 1.7$$

Applying Carson's Rule gives the RF bandwidth:

$$BW = 2 \times 3\,\text{kHz} \times (1.7 + 1) = 16.2\,\text{kHz}$$

Repeater frequencies are at least 15 kHz apart, with repeaters sharing reception areas separated by 30 kHz to reduce interference.

Those numbers apply to the RF signal sent to the antenna and, when scaled by the radio's frequency multiplier chain, also apply to the CE's output. For example, the 51.140 MHz repeater I mentioned earlier would transmit at 51.640 MHz from a 15.460000 MHz CE. To produce 5 kHz deviation at the transmit frequency, the CE's deviation Δf will be:

$$\Delta f = \frac{5\,\text{kHz}}{3} = 1.667\,\text{kHz}$$

Because the audio modulation bandwidth f_m remains 3 kHz, the CE's modulation index β: drops to 0.56 for a bandwidth of 9.4 kHz.

Homework: Use β to find the actual sideband amplitudes from the Bessel function and compare with Carson's Rule.

Hint: The sidebands extend well beyond the usual tabulations of J_0 and J_1.

SPEED VS. VOLTAGE VS. POWER

Although the AD9851 chip on the DDS module claims a 2.7 V to 5.5 V power supply

FIGURE 1
The PJRC Teensy 3.6 controls the DDS module on the right, with a 74AHCT125 level-shifting buffer converting the Teensy's LVCMOS SPI outputs to the AD9851's surprisingly high 3.5 V minimum VIL. The three circuits draw about 200 mA from the 5 V linear regulator on the heatsink.

range, the datasheet shows it requires a 5.0 V supply to run at 180 MHz from the 30 MHz clock oscillator installed on the module. If your application will allow a 30 MHz DDS clock, you can disable the chip's internal 6x PLL multiplier and run it at 30 MHz from a 2.7 V supply.

The Teensy 3.6 routes either 5 V from its USB cable or 3.6 to 6.0 V from its V_{IN} pin to an on-board 3.3 V linear regulator for its Arm Cortex M4 CPU. Because it must eventually run inside a repeater with no USB connection, I cut its USB power jumper and fed 5 V to the V_{IN} pin. You must not apply more than 3.3 V to any other pin and, of course, its logic outputs will not rise above 3.3 V.

Unfortunately, an AD9851 with a 5 V supply has a 3.5 V V_{IH} threshold. The 74AHCT125 Quad Bus Buffer parked between the two modules presents a 2.0 V V_{IH} to the Teensy and drives 4.4 V V_{OH} to the AD9851. **Figure 2** shows the connections between the module's chips.

The 5 V supply comes from an ON Semiconductor KA278RA05 linear regulator on the heatsink in Figure 1, providing the 200 2V a 2.7 V supply.

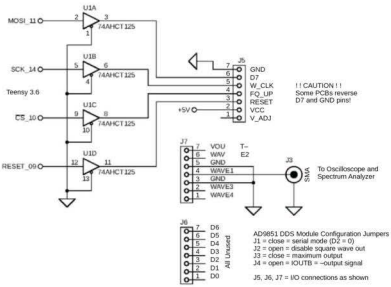


FIGURE 2
A 74AHCT125 Bus Buffer boosts the Teensy's LVCMOS SPI outputs to meet the AD9851's threshold requirements.

The 5 V supply comes from an ON Semiconductor KA278RA05 linear regulator on the heatsink in Figure 1, providing the 200 mA drawn by the other circuits. The regulator's 0.5 V dropout lets it run from a 6 V bench supply, far below the 9.6 V to 12.8 V available to Channel Elements inside various radios, but a linear regulator dissipating more than 1 W inside a CE-size case may justify a more complex power supply.

Producing the FM signal requires setting the DDS frequency by computing and sending 5 bytes for each audio sample. Setting the Teensy SPI hardware to use a 16 MHz SCK transfers each byte in half a microsecond and, with some software overhead, sends all 5 bytes in 3.5 µs. **Figure 3** shows the SCK signal shifting each data bit into the DDS input register. Because the SCK frequency falls into the same range as the CE oscillator frequencies, the firmware may need to adjust the SCK frequency to avoid interference with the DDS output.

FIGURE 3
The Teensy SPI clock signal ticks along at 16 MHz, well into the RF spectrum. Capturing a clean waveform required installing a grounding pin on the scope probe's tip, rather than using the relatively long alligator clip. Trace D3 presents the same SCK signal through the scope's digital pod, with the analog signal deskewed by -12 ns to align the edges.

The Teensy 3.6 ADC inputs have the same 0 to 3.3 V range as its logic inputs, which means the audio signal must be biased around $V_{CC}/2$ = 1.65 V and amplified to 3.3 V_{PP} to use the entire ADC range. As you should expect, different radios present different audio signal ranges to their Channel Elements, so the audio input will eventually require an amplifier with adjustable gain. I used test signals from a function generator biased at $V_{CC}/2$ and fed to the Teensy ADC through the gray wire in Figure 1.

```
void adcD_isr(void) {

double Audio;

   digitalWriteFast(PIN_ANALOG,HIGH);

   AnalogSample = adc->readSingle();            // fetch just-finished sample
   Audio = (AnalogSample - 2048) / 2048.0;      // convert to AC signal around zero

   DDSBuffer.Phase = 0x01;                      // phase = 0, no power-down, 6xREFCLK enable

   DDSBuffer.DeltaPhase = (uint32_t)((Audio * Deviation + Crystal - Offset) * CountPerHertz + 0.5);

   SPI.beginTransaction(SPISettings(16000000, LSBFIRST, SPI_MODE0));
   digitalWriteFast(PIN_DDS_FQUD, LOW);

   SPI.transfer((uint8_t)DDSBuffer.DeltaPhase);  // LSB first!
   SPI.transfer((uint8_t)(DDSBuffer.DeltaPhase >> 8));
   SPI.transfer((uint8_t)(DDSBuffer.DeltaPhase >> 16));
   SPI.transfer((uint8_t)(DDSBuffer.DeltaPhase >> 24));
   SPI.transfer(DDSBuffer.Phase);

   SPI.endTransaction();                        // do not raise FQ_UD until next timer tick!
... snippage ...
   digitalWriteFast(PIN_ANALOG,LOW);
}
```

DDS CALCULATIONS

As you saw in my September 2017 (*Circuit Cellar* 326) article, DDS frequency calculations and calibrations require numeric values with more significant figures than the standard AVR-based Arduino compiler could provide. The LF Crystal Tester firmware used 64-bit fixed-point arithmetic with 32-bit integers representing the integer and fractional parts of the value. Implementing those calculations on an Arduino Nano, however, definitely showed the limitations of its 8-bit microcontroller.

The Teensy 3.6 is "Arduino compatible" and uses the Arduino IDE, but it has a 32-bit Arm CPU running at 180 MHz, a hardware floating point unit, 256 kB of RAM, and a megabyte of Flash program memory. A quick test showed its C++ dialect implements true double precision floating point numbers with just under 16 significant figures of accuracy, sufficient to adjust VHF frequencies in millihertz steps. Even though I liked my fixed-point numbers, using native data types and operators makes much more sense.

The Arduino code excerpt in **Listing 1** runs at the completion of each ADC operation. You can download the complete source file from the *Circuit Cellar* article code webpage. The code reads the just-converted audio sample from the hardware, converts the 12-bit integer to a floating point value between ±1.0 representing the amplitude relative to the full-scale audio range, calculates the output frequency and the corresponding DDS delta-phase value, then sends it to the DDS.

The line converting the ADC sample to a double-precision floating-point value reveals an interesting section of the C language specification:

```
Audio = (AnalogSample - 2048) / 2048.0;
```

The `AnalogSample` variable is a 32 bit signed integer, twice the size of an AVR-based Arduino's 16 bit integer, with the ADC value

LISTING 1

When the ADC completes a conversion, it triggers this interrupt handler to convert each audio sample into the delta-phase value required to produce the corresponding frequency from the DDS. The SPI transaction configures the hardware to match the Little-Endian bit order used by the AD9851 DDS.

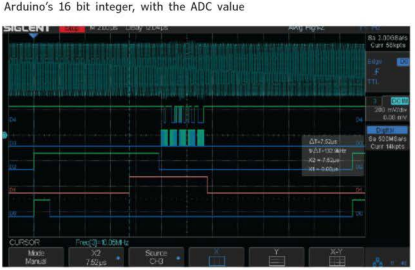


FIGURE 4

Pulses in D0 mark the 40 kHz ADC sampling rate, with the rising edge in D1 showing the ADC end of conversion interrupt. The FM calculations occur between that edge and the bursts in D3 and D4 transfer the result to the DDS. The rising edge in D2 latches the data from the previous conversion in the DDS registers. The analog trace at the top shows the 10 MHz DDS sine wave output.

LISTING 2
This timer interrupt handler runs every 25 µs to latch the DDS data produced by the previous ADC conversion cycle, then start another ADC conversion. The DAC output provides a visible indication of the ADC input.

```
void AudioSamplerIRQ(void) {

digitalWriteFast(PIN_TIMER,HIGH);

   digitalWriteFast(PIN_DDS_FQUD,HIGH); // latch previously shifted bits
   adc->startSingleRead(PIN_AUDIO, ADC_0); // start ADC conversion
   analogWriteDAC0(AnalogSample);        // show previous audio sample

digitalWriteFast(PIN_TIMER,LOW);
}
```

stored in its low 12 bits. Because the division involves a floating point constant, the compiler converts the integer result of the subtraction into a single precision `float` before doing the division in the floating point hardware unit. After the division, the compiler converts the result into a `double` and stores it in the `Audio` variable.

Casting the divisor into a double precision floating point number doesn't look much different:

```
Audio = (AnalogSample - 2048) / (double)2048.0;
```

However, the compiler must then perform a double-precision division adding nearly half a microsecond to the computation: about 90 instructions for a 180 MHz Arm CPU.

The line computing the DDS delta-phase value requires double precision arithmetic, because the various terms combine a

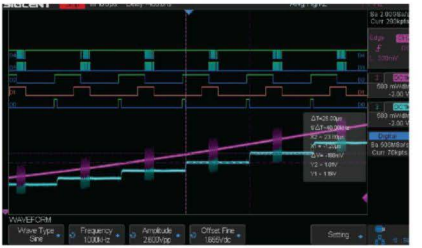


FIGURE 5
The firmware produces a "comfort signal" on the DAC output to show the ADC sampling works as intended. It also shows the DAC's inherent 25 µs delay from the ADC input. The analog hash corresponding to the SPI data transfer comes from hand-wired connections without a good ground plane.

Additional materials from the author are available at: www.circuitcellar.com/article-materials

RESOURCES

Analog Devices | www.analog.com

ON Semiconductor | www.onsemi.com

PJRC Store | www.pjrc.com

CE `Crystal` frequency near 10 MHz with a `Deviation` under 5 kHz and an `Offset` of a few hertz. The final result, however, will fit within the confines of the unsigned 32 bit integer fed to the DDS through the SPI hardware interface.

Homework: Cast various terms to other data types and measure the timing differences.

Programming at the hardware level requires not only a correct *result*, but a result at the correct *time*.

INTERRUPT SEQUENCING

As you saw earlier, amateur radio voice has a 3 kHz bandwidth, but some repeater installations now use control tones slightly above the normal voice frequencies, so a 4 kHz bandwidth seems prudent. The minimum sampling rate would thus be 8 kHz, with a higher rate simplifying the anti-aliasing filter required to eliminate higher frequencies. I decided to start by sampling the audio at 40 kHz rate, about five times the Nyquist rate, simply to determine whether such a pace was feasible, then adjust it to match reality.

Figure 4 shows the progress of a single audio sample through the hardware and firmware, with, somewhat to my surprise, plenty of time left over. The pulses on trace D0 (bottom), mark the audio sample timer interrupts occurring every 25 µs. The interrupt handler code in **Listing 2** latches the delta-phase bits sent to the DDS (falling edge of trace D2) after the previous ADC conversion, triggers the next ADC conversion, then copies the previous analog sample to a DAC output for monitoring. The PIN_TIMER outputs bracketing those lines produce the D0 pulse.

The ADC hardware provides control over the conversion bit width, its speed, and the number of samples averaged to produce the final result. The setup code configures the ADC hardware to average four successive 12-bit samples during the 7 µs conversion process, so each sample presented to the firmware already has some noise reduction.

The 6 µs pulse in trace D1 brackets the interrupt handler code in Listing 1, starting when the ADC triggers its sample-complete interrupt and ending after sending five bytes

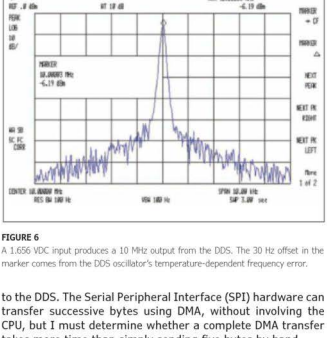


FIGURE 6
A 1.656 VDC input produces a 10 MHz output from the DDS. The 30 Hz offset in the marker comes from the DDS oscillator's temperature-dependent frequency error.

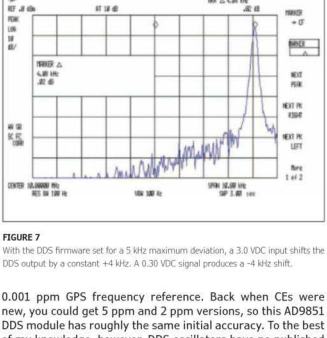


FIGURE 7
With the DDS firmware set for a 5 kHz maximum deviation, a 3.0 VDC input shifts the DDS output by a constant +4 kHz. A 0.30 VDC signal produces a -4 kHz shift.

to the DDS. The Serial Peripheral Interface (SPI) hardware can transfer successive bytes using DMA, without involving the CPU, but I must determine whether a complete DMA transfer takes more time than simply sending five bytes by hand.

The timer and ADC interrupt handlers occupy slightly less than 8 µs of the 25 µs between successive audio samples, accounting for 30% of the available CPU cycles. A CE has no "user interface," but this one needs a simple command parser to set the nominal frequency and deviation for a particular repeater, then store those values in EEPROM and fetch them during startup. With those functions in mind, the firmware tucks a simple cooperative multitasking supervisor into the available cycles.

The purple trace in **Figure 5** shows a 1 kHz 2.6 V_{PP} sine wave applied to the ADC input, with the corresponding DAC output in the cyan trace. The vertical cursors mark ADC samples and the horizontal cursors show the corresponding DAC outputs, with their intersections showing how each ADC sample appears in the next DAC output. The fuzz on the analog signals during the SPI data transfers reveals the inadequacy of a hand-wired proto board carrying mixed-signal RF circuitry.

FM DDS!

The radio multiplies the CE or DDS output's frequency before sending it to the transmitter's final RF amplifiers and antenna jack. Demodulating the RF back to the baseband signal through an FM receiver would provide an end-to-end check but looking at the DDS output on a spectrum analyzer must suffice until I acquire more radio hardware.

The spectrum in **Figure 6** comes from the DDS with a constant 1.656 VDC applied to the ADC input and the "crystal" frequency set to 10.000 MHz. As you saw earlier, the code in Listing 1 puts the input signal's AC zero voltage at the ADC's mid-span $V_{CC}/2$ level, so applying that voltage to the ADC produces a 10.000 MHz output with no frequency deviation.

Applying 3.0 VDC simulates the positive peak of a 2.7 V_{PP} sine wave around $V_{CC}/2$ and, with the firmware's `Deviation` variable set to 5 kHz, the signal in **Figure 7** appears at the expected 10.0 MHz + 4 kHz. Similarly, a 0.3 VDC input simulating the negative peak of the same sine wave puts the DDS output at 10.0 MHz - 4 kHz.

The marker in Figure 6 sits at 10.0 MHz + 30 Hz, a +3 ppm offset, as determined by my HP 8591 spectrum analyzer locked to a 0.001 ppm GPS frequency reference. Back when CEs were new, you could get 5 ppm and 2 ppm versions, so this AD9851 DDS module has roughly the same initial accuracy. To the best of my knowledge, however, DDS oscillators have no published tolerance or accuracy specifications, so this one may be unnaturally accurate.

The thermistors and capacitors visible to the left of the CE's crystal case in the lead photo reduced the effect of temperature variations, without the need for a stabilized crystal oven. My experience with the DDS oscillator in the LF Crystal Tester suggests the firmware can perform a similar correction by reading a temperature sensor attached to the oscillator case, with the DDS module tucked inside an insulating cover to prevent rapid fluctuations.

Applying sine, square, and noise signals to the DDS, in addition to the DC tests, show that the simple hardware in Figure 1 produces a reasonable frequency modulated signal that might be good enough to replace unobtainable Channel Elements. The signals definitely show the need to pay more attention to RF design rules before declaring victory, but I expected that.

CONTACT RELEASE: NORMALLY OPEN

Four decades and 200 articles after Steve Ciarcia asked me to write a regular column for his new *Circuit Cellar INK* magazine, this will be my last column. You'll find a continuing series of shop notes, projects, and curiosities on my blog at softsolder.com.

I'll see you along the Bit Stream!

ABOUT THE AUTHOR

Ed Nisley is an EE and author in Poughkeepsie, NY. Contact him at ed.nisley@pobox.com with "Circuit Cellar" in the subject line to avoid spam filters.